



EDTN Network

LOGIN / REGISTER

EE Expert Robert Ashby

Embedded Engineering

 SEARCH
 SEARCH
 SEARCH
 SEARCH
 SEARCH

SITE NAVIGATOR

Customize

Knowledge Centers ▶

Product Reviews

Data Sheets ▶

Guides & Experts ▶

News ▶

Network Sites ▶

International ▶

Ask Us

Circuit Cellar

Online Tools

App Notes

NetSeminars

Careers

Resources ▶

FAQ

[Archive](#) | [Main EE Expert](#) | [Guides & Experts](#)

Responses to "Menu Structures" Article

by [Robert Ashby](#)

My thanks to all of you for the many responses that I received on ways to store menus and user interfaces in small micros. I appreciate the feedback. It definitely helped to expand my knowledge on how other people would handle this particular situation. I would like to mention a few of the responses here.

First, Eric Earnst sent a link to www.eagleairaustr.com.au/sampcode.htm. They have some sample code here that help with various tasks. Towards the bottom of the page a "Table-Driven User Interface" was mentioned that was to be posted soon. I checked the link just before submitting my article for publication, but the interface has not yet been posted.

This particular application is supposed to store the interface in a serial EEPROM, and implements a 4 × 20 LCD display. The savings of ROM to implement the interface boasted a savings of over 50%. I look forward to reading this application note once it's been posted. Thanks, Eric, for the link.

Celeste Wood suggests using Microsoft Access sample databases to see how they set up switchboards. Even though this doesn't necessarily help me with the mechanics of storing the menu structures, they do give me a good view on how I might set up an easy-to-use interface. Thanks Celeste.

Bill Burton gave me a very good description of how he tackled a similar problem some years ago. I won't even attempt to paraphrase his message. [Click here](#) to read his response in its entirety.

Thanks, Bill. This is very useful information. The concept of using pointers is repeated in a few responses that I received. One thing that I really like about the structure that Bill mentions are the pointers to other adjacent menu items. They allow you to not be so concerned about the overall size and scope of all the menu items, you just need to look to see where you can go from your current location. Using pointers to the display string and action that a menu item takes allows you to store the actual messages and actions outside of the structure of the menu, and allows you to double-up messages and actions with different menu items without duplicating those messages and actions.

I felt that the pointers would allow me to store all the menu items back to back without any wasted space. They do add a little complexity. The biggest drawback that I could see is that if I wanted to replace an item of the menu that was in the middle, then I would have to change the entire menu storage since all the

 Test
 for
 con
 CD
 GPI
 UM

 Te
 2/1
 NetS
 for
 REG
Cry
MoKey
MoCry
Eng

following pointers would be affected.

Kenneth Reighard sent in an [excellent response](#) that pointed out some of the advantages of using a fixed-length array of strings for menu descriptions. He mentions that fixed-length menu items can be the way to go, especially if you are using an LCD module to display the menu. This way you can change the text of a menu item without affecting the other items in the list. If you need to add or delete a menu item, then you may need to affect the rest of the items in the menu. Kenneth also points out the possibility of storing menu indexes in nibbles rather than bytes to save some space if you're in a situation where it will help.

Aubrey Kagan also contributed her experience in the subject. She notes that it can be hard to generalize the exact implementation of such a task since resources are limited and you're never really sure what kind of hardware setup the end developer will have. She coded her projects in C and also noted to be careful not to squash the code too much as to hamper flexibility.

Aubrey's projects that she refers to use a 20×2 or a 20×4 LCD display and a setup of 4 buttons to run the interface: **Up**, **Down**, **Enter**, and **Menu**. The items on her menus need to allow multiple languages, various number formats, active manipulation of numbers on the screen, and a length of each menu item that actually exceeds the display.

The variables used in Aubrey's project are stored in a 256×16 EEPROM. Each variable is 16 bits in length. Similar arrays are set up in ROM that specify the minimum, maximum, and default settings for each variable. The messages are stored in ROM using a structure that identifies the message number, any process that needs to be performed on that message, the variable index that corresponds to that message, and the text of the message. The variable is displayed in place of the "@" character in the message. This assists Aubrey with the variance in placement of the value arising from language differences. This was very similar to a method that I used to handle the text messages on some alphanumeric LED displays on some fitness equipment in the World of Robert. I recommend that you read [Aubrey's response](#).

Your responses have been most helpful. My menu-driven project is already under way. I appreciate your help in the matter. I am using the characteristics of the methods that you suggested. You all have wonderful ideas and much knowledge to share. Please be sure to e-mail me with anything else that you would like to share. I will try to post what I can.

Embedded Engineering Archive

[Guides and Experts](#) [Analog Avenue](#) [EDA Tools](#) [PLD](#) [DSP](#) [EDA](#) [Embedded Systems](#) [Power](#) [Test](#)



[E-mail This Article](#)



[Printer-Friendly Page](#)

Copyright ©2002 ChipCenter-QuestLink

[About ChipCenter-Questlink](#) [Contact Us](#) [Privacy Statement](#) [Advertising Information](#) [FAQ](#)

The Center of the Electronics Universe
ChipCenter.

 Knowledge Centers

 Mfr Data Sheets

 Product Reviews

CoolRunner-II
High performance and ultra low power ...

Does your partner
meet your needs?


Super
Search
[Exa
Search
[LM3

 [E-mail This Article](#) [Printer-Friendly Page](#)

Copyright ©2002 ChipCenter-QuestLink

[About ChipCenter-Questlink](#) [Contact Us](#) [Privacy Statement](#) [Advertising Information](#) [FAQ](#)



Microchip Is



EDTN Network

LOGIN / REGISTER

SITE NAVIGATOR

Customize

Knowledge Centers ▶

Product Reviews

Data Sheets ▶

Guides & Experts ▶

News ▶

Network Sites ▶

International ▶

Ask Us

Circuit Cellar

Online Tools

App Notes

NetSeminars

Careers

Resources ▶

FAQ



Robert:

I, too, have been troubled by the lack of a some form of reference in the construction of menu structures on embedded systems. I will try to get my thoughts down in some kind of logical order and hope that it is comprehensible. Please don't hesitate to e-mail or phone me if you want to discuss this further. I have not tried to proofread this, and I hope it makes some sense.

Regarding sources of information about user interfaces and menu design. I recall some articles (many years ago!) in either *Communications of the ACM* or *ACM Computing Surveys* dealing with user interactions using text-based menus. A university library is likely to have these publications. IEEE Computer-Society publications are also a likely source. If I have a chance, I'll review my notes at home for pointers to some articles.

I did a project back in the MS-DOS 3.x days where I needed a text-based menu in a graphics environment. It needed to be fast, simple, and maintainable (we kept changing the instrument that was being controlled). What I came up with was crude, but it worked for many years. Basically, each menu item consisted of several pointers (they can be implemented also as "indexes"), one each:

- pointer to the next-right item on the current level
- pointer to the next-left item on the current level
- pointer to the first item in a sub-menu (or NULL) if no submenu
- pointer to the "action routine" for this item (or NULL)
- pointer to the display-string for this item.

Traversal of the menu tree was controlled by keyboard inputs:

<right-arrow> == next item right on current level

<left-arrow> == next item left on current level

<enter> == do action or go down one level

<down-arrow> or <page-down> == go down one level;
do nothing if no submenu

<up-arrow> or <page-up> or <escape> == go up one level;
do nothing if at top

<home> == return to top level

A stack was kept so completing an action or going up a level took you back to the

SEA
Sear
Exa
Sear
LM3

How
stuc
mic
rela

Te
2/1
NetS
for
REG

Cry
Mo

Key
Mo

Cry
Eng

place you came from.

Three display lines were used in place of drop-down menus:

1. The path you were on (e.g., **File > Save**)
2. The current menu-level selections with one item highlighted
3. The available submenu items for the highlighted item

My personal preferences for interface characteristics are:

- Commonly used items near the top
- Dangerous items separated from others (e.g., **File > Delete** should NOT be next to **File > Save**)
- Locations of items should not change (e.g., if **File > Save** is the 3rd option, it should stay the third option; use no-op placeholders if you know something additional will be added.

I hope this is of some use to you.

Sincerely,
Bill Burton

Close Window

[Guides and Experts](#) [Analog Avenue](#) [EDA Tools](#) [PLD](#) [DSP](#) [EDA](#) [Embedded Systems](#) [Power](#) [Test](#)



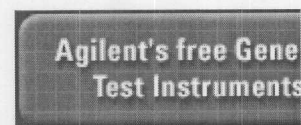
[E-mail This Article](#)



[Printer-Friendly Page](#)

Copyright ©2002 ChipCenter-QuestLink

[About ChipCenter-Questlink](#) [Contact Us](#) [Privacy Statement](#) [Advertising Information](#) [FAQ](#)



EDTN Network

LOGIN / REGISTER

Robert,

Your project is a perfect illustration of the multidisciplinary nature of engineering projects these days. The exercise is really one in computer programming rather than electronic engineering. Since EE and computers go hand-in-hand, many universities are linking their EE and computer engineering/computer science departments; current EE students seeking to get a leg up on the competition may do well to take comp eng courses.

Now to some possible solutions. While the file-allocation-table idea is workable, the FAT, or index, will take a good bit of space. Another idea would be to store the menu item descriptors as a string array. Languages such as BASIC store the strings as variable-length entities (the first byte gives the length), which would make updating easier. However, a fixed-size string (fixed at the display width) will make the storage and retrieval routine simpler, even though some storage bytes will be wasted if the string is shorter than the allocated space. Let's say the display is one of the common 16-character by 1-line LCD units. Each element of the string array will be 16 bytes long; this will accommodate item descriptors up to the full display width. With fixed string length, any descriptor may be retrieved by pointing to it with an index. The address of the first byte of a given string would be at $\text{tablebase} + 16 \times \text{index}$; the bytes could be read out for display easily just by reading 16 bytes starting at the address of the first byte. Tablebase is the starting address in memory of the string array for the particular menu; each menu level will have its own array. Index is a pointer, initially set to zero when the menu is first displayed with the pointer pointing at the first item. Index is incremented by one with each down-arrow button press to point to the next menu item (decremented by one with each up-arrow button press to point to the previous item.) Logic would have to be included to prevent index from exceeding the number of items in the menu or decreasing below zero.

Index is also used to point into a jump table (or select structure) to determine the next action to be taken by the software. If a menu item is an action, the jump table entry corresponding to that index value is the starting address of a subroutine to take that action. If a menu item is a submenu, the jump table entry is to the menu-display routine. (One routine can display all menus; but if there are many menu levels stack space may become a problem due to the recursive subroutine calls.)

Another array of byte-length integers would be used to keep track of where the user is in the menu tree. $\text{menulevel}(0)$ would be the top level; $\text{menulevel}(N-1)$ would be the bottom level of an N-level menu tree. The value stored in $\text{menulevel}(i)$ would indicate which menu or submenu was selected or entered. The rest of the program would then know how the menu was navigated, if it needed to know.

If there are no more than 16 menu choices at each level, the $\text{menulevel}()$ array could be compacted by packing two nibble-length integers into each byte. This would cut the array size in half, but at the expense of some additional code required to pack and unpack the bytes. (Some microprocessors have instructions for packing and unpacking BCD digits; these may possibly be used as BCD digits take only four bits. With such a processor, the overhead for packing and unpacking is reduced.)

SEA
Sear
[Exa
Sear
LM3

Ret
and
ther
Onl
Des

Te
2/T
NetS
for
REG

Cry,
Mo

Key
Mo

Cry,
Eng

String array space could be saved by limiting each menu item descriptor to less than 16 characters; each element of the string array would be correspondingly shortened.

It is difficult to provide a specific answer without knowing the details of the case. For example, there is only 1 Kbyte available for the menu structure, but how many menus (and how many menu items) are there? It may be that the arrays will have to be "packed," that is, each element will be variable length and the next element will begin at the first byte following the last byte of the previous element. In this case, there will be substantial programming overhead to be able to locate the first byte of each string to display it. Furthermore, any change to any item which changes its length will require all of the elements following it to be moved. With the fixed-length strings, the only time moving of blocks of string data would be required is if more items were added to a menu, increasing the number of strings in its array. Changes to existing menu items would not require other items to be moved to accommodate the change.

I hope these ideas are helpful.

Sincerely,
Kenneth W. Reighard

Close Window

[Guides and Experts](#) [Analog Avenue](#) [EDA Tools](#) [PLD](#) [DSP](#) [EDA](#) [Embedded Systems](#) [Power](#) [Test](#)



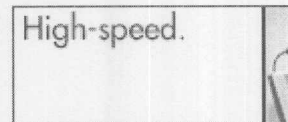
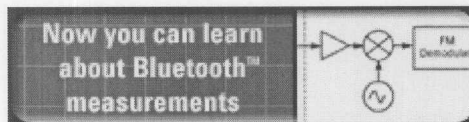
[E-mail This Article](#)



[Printer-Friendly Page](#)

Copyright ©2002 ChipCenter-QuestLink

[About ChipCenter-Questlink](#) [Contact Us](#) [Privacy Statement](#) [Advertising Information](#) [FAQ](#)



EDN Network

LOGIN / REGISTER

Robert-

SITE NAVIGATOR

Customize

Knowledge Centers ▶

Product Reviews

Data Sheets ▶

Guides & Experts ▶

News ▶

Network Sites ▶

International ▶

Ask Us

Circuit Cellar

Online Tools

App Notes

NetSeminars

Careers

Resources ▶

FAQ

VOLTAGE
REGULATORS

Commodities or
Innovative Devices
ST has the solution

I, too, have been troubled by the lack of a some form of reference in the construction of menu structures on embedded systems. I will try to get my thoughts down in some kind of logical order and hope that it is comprehensible. Please don't hesitate to e-mail or phone me if you want to discuss this further. I have not tried to proofread this, and I hope it makes some sense.

Despite having penned a number of articles, design ideas, and application notes, I still find it difficult to explain these software concepts. I hope there is something here that you can use. I have done several systems like this, and each time it is a little different, and so I don't really have any consistency. When I started my second project in this vein, I discounted the idea of an article, given the possible variations in hardware implementation and project requirements whether it is possible to generalize enough to make the implementation formulaic. If, however, you think that there is a literary future to these concepts, perhaps we could collaborate on the article.

Although I generally prefer to code in Assembler, I would hate to have tried to implement this in a low-level language. I hope you are familiar with C. Also I am not sure what part of this you want to appear in 1K of memory, but I have my doubts that it will fit with a generic approach. Based on my experience, a specific approach intended to save memory will severely cramp your style when modifications are requested.

All of my projects that use this are implemented with a simple 4-button keyboard (**Up**, **Down**, **Enter**, and **Menu**), and a simple LCD display—I have used a 20 × 2 and 20 × 4 at different times.

The keyboard philosophy is to start the access to the menu structure from the background display, then press *Menu*. Use the **Up** and **Down** buttons to select your choice, and press **Enter** to finalize (go down a level or modify a parameter). **Menu** will then take you up a level in the menu structure. In other words, aside from the first step, **Enter** goes down a level, **Menu** goes back up—I know this seems inconsistent, but the users don't seem to find it counter-intuitive.

The purpose of the layered structure is to display variables and constants, and to modify the constants. For what it's worth, I found the smaller screen easier to work with because there is only one option on the screen at a time, so the process of what to display was sequential. With 4 lines, there would be 4–8 lines of options in a menu, and scrolling them up and down became complicated. If I remember, I will come back to how I achieved this later.

Problems that I have that make life more difficult:

- Multiple languages make placement of a variable within text different from language to language. In a simple example, in English the word HUMIDITY is 8 letters, these would followed by 4 spaces for a 3-digit humidity number followed by the "%" symbol. In German, the word is DAMPFLEISTUNG, so the placement of the variable would be much

further to the right.

- The constants that can be changed are inconsistent. Sometimes they are ON or OFF (and 1 or 0 doesn't cut it with the user), other times they are a number 1–100 or 0–255, and sometimes 0–65535 (modifying the values can take time). In other instances, it is a password that is entered one digit at a time, and is not treated as a 4-digit number. Time and date are also different in that there are a number of variables simultaneously on one line, e.g., month/date/year.
- The level in the hierarchy where a parameter can be set will vary depending on how many levels you have to go through to get to it.

All my constants were stored in a serial 256 × 16 EEPROM, and so I based all my constants as a 16-bit number, even if the only possible values were 0 and 1.

The variables are created as one long list and given a number associated with the location in EEPROM, e.g.,

```
#define UPPER_BLOWER_SPEED 4 //value stored at address 0x4 in eeprom
#define LOWER_BLOWER_SPEER 5 // value stored at address 0x5 in eeprom
```

I created 3 arrays in ROM with 256 **int** elements each. One array defines the lowest value a parameter may have, the second defines the highest value a parameter may have, and the third defines the default value on initial power-up. (Actually there are fewer than 256 elements since there is a check sum to determine the integrity of the data.) Changing the numbers can only be done as **Up** and **Down**, so each time the number is modified it is compared to the upper or lower limit. Holding the key down for an extended period results in the increment being changed from 1 to 10, then 100, and finally 1000. Unfortunately there are some parameters (like the aforementioned ON and OFF) that this doesn't apply to, and I do not yet know how one would generalize this to create a simple, short software module to do it.

I initially created a list of messages in ROM associated with the parameters, and tried to get the customer to scroll sequentially through these—so there was now hierarchy. This worked well, but the users and/or marketing decided that an hierarchical approach would be better, so the order was immediately scrambled. This appears to be par for the course for any user interface, so ease of update is paramount.

My approach has been to create a message in RAM using constants in ROM plus all the other massaging that is needed before writing it to the display. That means that there could be a 10-line message intended for the display, with the 4 lines of the display acting as a "window" to this display. This makes moving a selection up or down much easier.

Each message is of fixed length, and is created as a constant array. Associated with each message is a sequential number as an ID. They can be entered in any order, so that updating is not a problem. If a variable is to be inserted in each line, I place a special character where I want the variable to appear (to cater for the variable length of expressions in different languages). I have reserved the "@" for this purpose.

I created a structure

```
struct ScreenType {
```

```

    unsigned char NumberOfLines;
    unsigned char MsgNo[MAX_SCREEN_SEL]; // message to be displayed
    unsigned char ProcNum[MAX_SCREEN_SEL]; // process to be done on message
    unsigned char ParamNum[MAX_SCREEN_SEL]; // parameter associated with line
    // this can carry a number associated with the line, so that the lines can
    // be packed and still correctly identified (it could be eeprom address
    // when the line is actually a parameter
}

```

and then invoked it with

```

struct ScreenType const Screen[70]={
    4, // 0-Initial Display=4 lines
    0,0,0,0,0,0,0,0,0,0, // message number
    1,2,3,4,0,0,0,0,0,0, // processes
    0,0,0,0,0xff,0xff,0xff,0xff,0xff,0xff, // associated parameters

    5, // 1-Menu display
    7,8,9,10,70,0,0,0,0,0, // message number
    0,0,7,8,9,0,0,0,0,0, // processes
    0,1,2,3,4,0xff,0xff,0xff,0xff,0xff, // associated parameters
    .
    .
    .
}

```

From this you can see that there are 70 different screens that can be displayed (this is obviously application-dependent). The first number dictates the number of lines in the screen (as opposed to the 4 lines of the display). The MAX_SCREEN_SEL macro value is 10, meaning that there can be 10 lines per screen. The MsgNo parameter refers to the screen line ID. This means you can group up to 10 lines of predetermined messages to a screen. Associated with each line is a "process type" and a parameter address. A line can be a plain line with no change ever needed. It can be present or omitted (sort of equivalent to "grayed out" in Windows), and it can have a variable inserted. The process type will allow the implementation of these features. The parameter may allow generalization of the process so that fewer processes are needed. Keep in mind, as I said before, that there are many different kinds of variables that need different presentation on the display. This is how I managed to achieve it. Each line is copied to a screen buffer in RAM, and all the processes are executed on each line. If a line was to be removed by the process type, it is removed from the RAM image, and the subsequent lines would be moved up. Any variables that are to be displayed are processed to ASCII and placed in the line. The associated parameter moves along with each line. When this is complete, the display is enabled as a window onto the screen buffer.

This is all support to the actual implementation of the hierarchical menu structure. At the time that I wrote the software, I could not conceive of how to generalize this, and so each basic menu screen has a chunk of code associated with it. Some of the problems I had included slight variations in keyboard entry actions, e.g., setting up passwords, changing time/date, as opposed to a single variable. I thought I had the Holy Grail, but I was wrong when I came to implement it. As I look at this with hindsight, I think it may be possible by adding some additional information in the above ScreenType structure to allow a more general approach. One concept that I think is worth preserving is the "push/pop" idea. As I mentioned before, the **Menu** button takes you up a level in the hierarchy, so when the **Enter** button is pushed, save ("push") the current conditions of execution (where in the hierarchy you are) so that this may be reconstructed on a return. It may be possible to define a "map" as a constant multi-dimensional array with entries corresponding to the menu structure. Perhaps this is an area to explore.

-Aubrey Kagan

Close Window